фективной из всех трех алгоритмов прямой (строгой) сортировки, то следует ожидать относительно существенного улучшения. И все же это выглядит как некий сюрприз: улучшение метода, основанного на обмене, о котором мы будем сейчас говорить, оказывается, приводит к самому лучшему из известных в данный момент методу сортировки для массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар [2.5] и [2.6] даже назвал метод *быстрой сортировкой* (Quicksort).

В Quicksort исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, у нас есть n элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за n/2 обменов, сначала поменять местами самый левый с самым правым, а затем последовательно двигаться с двух сторон. Конечно, это возможно только в том случае, когда мы знаем, что порядок действительно обратный. Но из этого примера можно извлечь и нечто действительно поучительное.

Давайте, попытаемся воспользоваться таким алгоритмом: выберем наугад какой-либо элемент (назовем его x) и будем просматривать слева наш массив до тех пор, пока не обнаружим элемент $a_i > x$, затем будем просматривать массив справа, пока не встретим $a_j < x$. Теперь поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть, с ключами меньше (или равными) x, и правую — с ключами больше (или равными) x. Теперь этот процесс разделения представим в виде процедуры (прогр. 2.9). Обратите внимание, что вместо отношений $>$ и $<$ используются $\geqslant$ и $\leqslant$, а в заголовке цикла с WHILE — их отрицания $<$ и $>$. При таких изменениях x выступает в роли барьера для того и другого просмотра. Если взять в качестве x для сравнения средний ключ 42, то в массиве ключей

44 55 12 42 94 06 18 67

```
PROCEDURE partition;
  VAR w, x: item;
BEGIN i := 1; j := n;
  случайно выбрать x;
  REPEAT
    WHILE a[i] < x DO i := i+1 END;
    WHILE x < a[j] DO j := j-1 END;
    IF i <= j THEN
      w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
    END
  UNTIL i > j
END partition
```

Прогр. 2.9. Сортировка с помощью разделения.

для разделения понадобятся два обмена: 18 ↔ 44 и 6 ↔ 55

18 06 12 42 94 55 44 67

последние значения индексов таковы: $i = 5$, а $j = 3$. Ключи $a_1 \dots a_{i-1}$ меньше или равны ключу $x = 42$, а ключи $a_{j+1} \dots a_n$ больше или равны x. Следовательно, существует две части, а именно

$$
\begin{aligned}
&Ak: 1 \leqslant k < i : a_k \leqslant x \\
&Ak: j < k \leqslant n : x \leqslant a_k
\end{aligned}
\qquad (2.14)
$$

Описанный алгоритм очень прост и эффективен, поскольку главные сравниваемые величины i, j и x можно хранить во время просмотра в быстрых регистрах машины. Однако он может оказаться и неудачным, что, например, происходит в случае n идентичных ключей: для разделения нужно n/2 обменов. Этих вовсе необязательных обменов можно избежать, если операторы просмотра заменить на такие:

```
WHILE a[i] < = x DO i := i + 1 END;
WHILE x < = a[j] DO j := j - 1 END
```

Однако в этом случае выбранный элемент x, находящийся среди компонент массива, уже не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут, если только не использовать более сложные условия их окончания, к переходу через границы массива. Простота условий, употребленных в прогр. 2.9, вполне оправдывает те дополнительные обмены, которые происходят в среднем относительно редко. Можно еще немного сэкономить, если изменить заголовок, управляющий самим обменом: от $i \leqslant j$ перейти к $i < j$. Однако это изменение не должно касаться двух операторов: $i := i + 1$, $j := j - 1$. Поэтому для них потребуется отдельный условный оператор. Убедиться в правильности алгоритма разделения можно, удостоверившись, что отношения (2.14) представляют собой инварианты оператора цикла с REPEAT. Вначале при $i = 1$ и $j = n$ их истинность тривиальна, а при выходе с $i > j$ они дают как раз желаемый результат.

Теперь напомним, что наша цель — не только провести разделение на части исходного массива элементов, но и отсортировать его. Сортировку от разделения отделяет, однако, лишь небольшой шаг: нужно применить этот процесс к получившимся двум частям, затем к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются программой 2.10.

Процедура *sort* рекурсивно обращается сама к себе. Рекурсии в алгоритмах — это очень мощный механизм, его мы будем рассматривать в гл. 3.

```
PROCEDURE QuickSort;

  PROCEDURE sort(L, R: index);
    VAR i, j: index; w, x: item;
  BEGIN i := L; j := R;
    x := a[(L+R) DIV 2];
    REPEAT
      WHILE a[i] < x DO i := i+1 END ;
      WHILE x < a[j] DO j := j-1 END ;
      IF i <= j THEN
        w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
      END
    UNTIL i > j;
    IF L < j THEN sort(L, j) END ;
    IF i < R THEN sort(i, R) END
    END sort;

 BEGIN sort(1, n)
 END QuickSort
```

Прогр. 2.10. Quicksort.